Date=11/11/2020
Lecture By=Manish Mahant
Subject ⇒Functional Programing

| IN PREVIOUS LECTURE **(QUICK RECAP) Date-10/11/2020** | **In Today's Lecture (Overview)** |
|---|---|
| Iterators<br>Generator functions<br>yield<br>Syntax<br>Description<br>Questions For the Self-Practice | What Is Functional Programming??<br>Side Effects<br>Conclusion<br>Questions For Self-Practice |

# What Is Functional Programming??

Functional programming (often abbreviated FP) is the process of building software by composing pure functions, avoiding shared state, mutable data, and side-effects. Functional programming is declarative rather than imperative, and application state flows through pure functions. Contrast with object oriented programming, where application state is usually shared and colocated with methods in objects.

# Pure functions

Just because your program contains functions does not necessarily mean that you are doing functional programming. Functional programming distinguishes between pure and impure functions. It encourages you to write pure functions.
A pure function must satisfy both of the following properties:
**Referential transparency:** The function always gives the same return value for the same arguments. This means that the function cannot depend on any mutable state.
**Side-effect free:** The function cannot cause any side effects. Side effects may include I/O (e.g., writing to the console or a log file), modifying a mutable object, reassigning a variable, etc.
Let's illustrate with a few examples. First, the **multiply** function is an example of a pure function. It always returns the same output for the same input, and it causes no side effects.

# Immutability

Let's return to the concept of captured variables. Above, we looked at the **canRide** function. We decided that it is an impure function, because the **heightRequirement** could be reassigned. Here is a contrived example of how it can be reassigned with unpredictable results:

```
let heightRequirement = 46;

function canRide(height) {
  return height >= heightRequirement;
}

// Every half second, set heightRequirement to a random number between 0
and 200.
setInterval(() => heightRequirement = Math.floor(Math.random() * 201),
500);

const mySonsHeight = 47;

// Every half second, check if my son can ride.
// Sometimes it will be true and sometimes it will be false.
setInterval(() => console.log(canRide(mySonsHeight)), 500);
```

Let me reemphasize that captured variables do not necessarily make a function impure. We can rewrite the **canRide** function so that it is pure by simply changing how we declare the **heightRequirement** variable.

```
const heightRequirement = 46;

function canRide(height) {
  return height >= heightRequirement;
}
```

Declaring the variable with **const** means that there is no chance that it will be reassigned. If an attempt is made to reassign it, the runtime engine will throw an error; however, what if instead of a simple number we had an object that stored all our "constants"?

```
const constants = {
```

```
  heightRequirement: 46,
  // ... other constants go here
};

function canRide(height) {
  return height >= constants.heightRequirement;
}
```

We used **const** so the variable cannot be reassigned, but there's still a problem. The object can be mutated. As the following code illustrates, to gain true immutability, you need to prevent the variable from being reassigned, and you also need immutable data structures. The JavaScript language provides us with the **Object.freeze** method to prevent an object from being mutated.

```
// CASE 1: The object is mutable and the variable can be reassigned.
let o1 = { foo: 'bar' };

// Mutate the object
o1.foo = 'something different';

// Reassign the variable
o1 = { message: "I'm a completely new object" };


// CASE 2: The object is still mutable but the variable cannot be
reassigned.
const o2 = { foo: 'baz' };

// Can still mutate the object
o2.foo = 'Something different, yet again';

// Cannot reassign the variable
// o2 = { message: 'I will cause an error if you uncomment me' }; //
Error!


// CASE 3: The object is immutable but the variable can be reassigned.
let o3 = Object.freeze({ foo: "Can't mutate me" });
```

```
// Cannot mutate the object
// o3.foo = 'Come on, uncomment me. I dare ya!'; // Error!

// Can still reassign the variable
o3 = { message: "I'm some other object, and I'm even mutable -- so take
that!" };


// CASE 4: The object is immutable and the variable cannot be reassigned.
This is what we want!!!!!!!!
const o4 = Object.freeze({ foo: 'never going to change me' });

// Cannot mutate the object
// o4.foo = 'talk to the hand' // Error!

// Cannot reassign the variable
// o4 = { message: "ain't gonna happen, sorry" }; // Error
}
```

Hiiiii

# Function composition

Remember back in high school when you learned something that looked like **(f ∘ g)(x)**? Remember thinking, "When am I ever going to use this?" Well, now you are. Ready? **f ∘ g** is read "**f composed with g**." There are two equivalent ways of thinking of it, as illustrated by this identity: **(f ∘ g)(x) = f(g(x))**. You can either think of **f ∘ g** as a single function or as the result of calling function **g** and then taking its output and passing that to **f**. Notice that the functions get applied from right to left—that is, we execute **g**, followed by **f**.
A couple of important points about function composition:

We can compose any number of functions (we're not limited to two).

One way to compose functions is simply to take the output from one function and pass it to the next (i.e., **f(g(x))**).

```
// h(x) = x + 1
// number -> number
function h(x) {
  return x + 1;
}

// g(x) = x^2
// number -> number
function g(x) {
  return x * x;
}

// f(x) = convert x to string
// number -> string
function f(x) {
  return x.toString();
}

// y = (f ∘ g∘h)(1)
const y = f(g(h(1)));
console.log(y); // '4'
```

## Side Effects

A side effect is any application state change that is observable outside the called function other than its return value. Side effects include:

- Modifying any external variable or object property (e.g., a global variable, or a variable in the parent function scope chain)
- Logging to the console
- Writing to the screen
- Writing to a file
- Writing to the network
- Triggering any external process
- Calling any other functions with side-effects

Side effects are mostly avoided in functional programming, which makes the effects of a program much easier to understand, and much easier to test.

Haskell and other functional languages frequently isolate and encapsulate side effects from pure functions using **monads**. The

topic of monads is deep enough to write a book on, so we'll save that for later.

What you do need to know right now is that side-effect actions need to be isolated from the rest of your software. If you keep your side effects separate from the rest of your program logic, your software will be much easier to extend, refactor, debug, test, and maintain.

This is the reason that most front-end frameworks encourage users to manage state and component rendering in separate, loosely coupled modules.

## Conclusion

Functional programming favors:

- Pure functions instead of shared state & side effects

- Immutability over mutable data
- Function composition over imperative flow control
- Lots of generic, reusable utilities that use higher order functions to act on many data types instead of methods that only operate on their colocated data
- Declarative rather than imperative code (what to do, rather than how to do it)
- Expressions over statements
- Containers & higher order functions over ad-hoc polymorphism

## Questions For Self-Practice

https://au-assignment.s3.amazonaws.com/Week_20_Day_3_Assignment-edc23cab-d824-4a6f-90a0-3a8a78a4fbb7.pdf